

A Batch of PNUTS: Experiences Connecting Cloud Batch and Serving Systems

Adam Silberstein, Russell Sears
Yahoo! Research
Santa Clara, CA, USA
{silberst,sears}@yahoo-inc.com

Wenchao Zhou
U. Penn
Philadelphia, PA, USA
wenchaoz@seas.upenn.edu

Brian F. Cooper *
Yahoo! Research
Santa Clara, CA, USA
brianfrankcooper@yahoo.com

ABSTRACT

Cloud data management systems are growing in prominence, particularly at large Internet companies like Google, Yahoo!, and Amazon, which prize them for their scalability and elasticity. Each of these systems trades off between low-latency serving performance and batch processing throughput. In this paper, we discuss our experience running batch-oriented Hadoop on top of Yahoo!’s serving-oriented PNUTS system instead of the standard HDFS file system. Though PNUTS is optimized for and primarily used for serving, a number of applications at Yahoo! must run batch-oriented jobs that read or write data that is stored in PNUTS.

Combining these systems reveals several key areas where the fundamental properties of each system are mismatched. We discuss our approaches to accommodating these mismatches, by either bending the batch and serving abstractions, or inventing new ones. Batch systems like Hadoop provide coarse task-level recovery, while serving systems like PNUTS provide finer record or transaction-level recovery. We combine both types to log record-level errors, while detecting and recovering from large-scale errors. Batch systems optimize for read and write throughput of large requests, while serving systems use indexing to provide low latency access to individual records. To improve latency-insensitive write throughput to PNUTS, we introduce a batch write path. The systems provide conflicting consistency models, and we discuss techniques to isolate them from one another.

Categories and Subject Descriptors: H.2.4 [Systems]: distributed databases

General Terms: Performance

1. INTRODUCTION

The recent proliferation of cloud data management systems has primarily produced two kinds of architectures: *batch-oriented* data analysis systems like Hadoop [3] and Dryad [16],

*Now at Google, Inc. Work done while at Yahoo! Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD’11, June 12–16, 2011, Athens, Greece.

Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...\$10.00.

and *online-oriented* serving systems like PNUTS [10], Cassandra [2], Dynamo [13], and others. Although both types of systems leverage massively parallel infrastructures and provide simplified data models and query interfaces, they are tuned for very different classes of workloads. Batch systems focus on high throughput for processing very large files, while serving systems focus on low latency access to relatively smaller records.

However, many interesting web applications do not fit neatly into either the serving or batch paradigm. For example, ad targeting systems store online information about user actions, and use batch jobs to analyze these actions. Similarly, social networking sites use batch processing to make recommendations and filter spam from content posted in real-time by users. A variety of other applications are based on this *hybrid processing* model, where low latency reads and writes are needed to serve web pages and track activity, and high throughput batch operations are used to analyze this activity and generate more pages to serve.

Yahoo! is a heavy user of both PNUTS (online), and Hadoop (batch). Although these systems were initially developed separately, it has become clear that many Yahoo! applications would benefit from a principled architecture that combines the two systems. In fact, in several cases, applications had built their own software to “glue” the two systems together, and we realized that it was far more effective to build a common set of abstractions to tie the batch and online systems together, despite their underlying architectural differences.

Though we focus on lower level details of two specific systems prominent at Yahoo!, PNUTS and Hadoop, the high-level design patterns we lay out can be broadly used to bridge the gap between batch and online-oriented systems. In this paper, we report on our experiences in combining a batch system with an online system. Our use cases so far can be classified in three patterns:

- **Load:** Applications frequently migrate data to PNUTS from legacy systems or load data gathered from other sources. They need a fast and failure-resistant way to move their data.
- **Update:** Applications perform offline analyses of user access logs and other data sets, and push the output to PNUTS. The main platform at Yahoo! for these analyses is Hadoop.
- **Scan:** Applications run Hadoop analyses that read from a live PNUTS table, whether as a one-off job, or as the start of heavier weight analyses.

In particular, we examine three areas where the fundamental properties of each system are mismatched, and where we had to either break the batch and serving abstractions to a degree or invent new abstractions:

- **Failure handling** - Batch systems provide task-level recovery: when a map task fails in Hadoop, the whole job is restarted. In contrast, serving systems provide record or transaction-level recovery: updates are re-applied to individual records as necessary.
- **Performance optimization** - Batch systems optimize for read and write throughput by organizing data into large blocks. For example, HDFS (and even "record oriented" batch systems like HBase) use log-structured (append-only) files. Serving systems employ indexing to provide access to individual records with low latency. Pnuts for example utilizes a traditional B-tree structure with update-in-place semantics.
- **Consistency** - Batch systems provide block-level, synchronous consistency designed for writes that happen in batches. In contrast, serving systems provide record level consistency, and record versions can temporarily diverge because of asynchrony.

These mismatches are apparent even in "simple" hybrid applications. For example, one major use case is to bulk load a Pnuts table from the results of a Hadoop job. When bulk loading terabytes of data, failures will occur; the speed of the bulk load depends heavily on how writes are optimized; and the correctness of the resulting Pnuts table depends on how consistency was enforced (especially if there are concurrent writes to the table being loaded).

Our goal is to facilitate data movement in both the batch-to-online and online-to-batch directions. We find that the mismatches outlined above primarily plague batch-to-online, so much of our attention focuses on that case.

Instead of trying to combine Pnuts and Hadoop, we could have designed and built a new system with a hybrid architecture. Alternatively, we could have built a custom system for each task instead of trying to develop a general solution. We chose to combine Hadoop and Pnuts for these tasks, despite the difficulties, for several reasons. First, these systems are mature and stable, and the amount of effort to develop abstractions which combine them should be significantly less than building a new system from scratch. Second, there are still many use cases where only one system (batch or serving) is needed, and thus Pnuts and Hadoop will continue to have independent existence. The engineering cost of maintaining three systems (Pnuts, Hadoop, and a new hybrid system) or of maintaining a custom, one-off use-case-specific system would be higher than the cost of maintaining a glue layer. Furthermore, we wanted to support applications that started on one system but expanded to a hybrid architecture later; for example a web application on Pnuts that later added Hadoop for analytics. This transition would be easiest if the system could stay on Pnuts, but begin to use the new capabilities of the hybrid layer.

This paper discusses the following contributions:

- We present the architecture, based on Hadoop connector classes, for Hadoop-on-Pnuts (Section 3).
- We discuss how we deal with the mismatch of failure models by allowing bits of each abstraction (batch and serving) to "leak" into the other (Section 4).

- We introduce our *snapshot-based* bulk write path that provides high throughput while minimizing impact on Pnuts' normal record-oriented write path (Section 5).
- We examine how the above techniques interfere with the consistency models of the systems, and discuss techniques for working around the mismatch to ensure correct data (Section 6).

Aside from the challenges we have mapped above, we detail the bulk load use case in Section 1.1, we also include in this paper a discussion of related work in Section 2, and an experimental evaluation of our approaches in Section 7.

1.1 Batch Job Example

We now detail a specific batch job case, which we use as a running example. Among the Pnuts adopters at Yahoo! are applications moving data from legacy systems to Pnuts. One of our key applications has several TB of data on 20-30 tables. These tables are live and constantly updated. Moreover, to minimize risk, the application wanted to not just switch from their legacy system "L" to Pnuts, but send updates to both, keeping L up-to-date and available as a standby. Since L is live, we only had access to a weekly backup, which we copied to HDFS. Though this task involves moving data from one serving system to another, a data load is a large batch operation, and thus our hybrid Pnuts/Hadoop architecture was well suited for the job.

Both L and Pnuts rely on a messaging layer for cross-data center replication; writes performed in one data center are transmitted to the others and applied there. We rely on this messaging layer for loading Pnuts and for keeping L up-to-date with it after the load is complete. The messaging layer lets a client declare a subscription to the writes on a table, and delivers those writes to the client one-by-one as the client requests them. The messaging layer buffers writes until they are consumed. We load table-by-table. Algorithm 1 details the per-table steps.

Algorithm 1 Bulk load from legacy data store.

1. Pnuts subscribes to the messaging layer for a table, but does not consume any writes. The messaging layer therefore starts buffering writes to the table.
 2. Copy the table from System L backup to HDFS.
 3. Using Hadoop-on-Pnuts, load L table from HDFS to Pnuts.
 4. Pnuts begins consuming buffered writes from the message layer and applying them.
 5. Clients divert live traffic to Pnuts, but L stays subscribed to the messaging layer and continues applying writes.
-

While it is possible to streamline Algorithm 1 further by reading data directly from the L backup, we write to HDFS to avoid creating a long data pipeline; writing to HDFS contributes little to overall running time.

Algorithm 1 illustrates the careful dance that must be followed to correctly move a large amount of live, concurrently updated data from one system to another. For example, updates applied to system L during the load must be buffered for later insertion into Pnuts, in case they were applied after the snapshot was constructed. Similarly, we proceeded table-by-table to minimize risk as well as minimize the amount of time each table was in a vulnerable

transition phase. However, it also means that for a long period, some tables had their source of truth in L and some had their source of truth in PNUTS. And even in the table-by-table approach, a high write rate for one table means a large amount of storage is required to buffer all of the updates.

This example also shows that combining multiple systems can introduce unexpected failure modes. L and PNUTS share the same internal record format, JSON. Thus, our bulk loading job does no data transformation, and is essentially a pass-thru. While this seemingly should minimize failures (any L record should successfully insert into PNUTS), in practice, most JSON libraries break standards compliance in various minor ways. PNUTS uses a newer, more standards-compliant JSON library than L, which leads to errors during bulk operations. We discuss handling such failures in detail in Section 4.

We implemented this bulk load using our hybrid Hadoop-on-PNUTS architecture. Bulk load scans the L snapshot with one or more parallel scanners, each responsible for some portion of the snapshot. Each scanner reads in records one-at-a-time from the snapshot, transforms those records into PNUTS requests, and loads them into PNUTS. This stage forces us to address the batch/serving mismatches we introduced in Section 1.

- **Failure handling/checkpointing:** There are several failure types that may occur during bulk load. Individual writes may fail and we must handle these and move on. Network partitions, etc. may cause massive failures and we must detect these and halt the load. On re-start, we want to continue the job where it broke, rather than from the beginning. Finally, Hadoop failures can occur, in which case we leverage Hadoop’s failure handling.
- **Speed:** The bulk load must be high throughput to complete quickly. At the same time, since PNUTS is a multi-tenant system serving live tables, we must be able to throttle loads to avoid saturating PNUTS.
- **Consistency model:** Bulk load must come with an explanation of the races it permits between bulk and standard operations, as well as a set of mechanisms for optionally preventing them.

Other use cases: The typical bulk update case is one where an application takes data sets that funnel into Hadoop (stored in HDFS), runs analyses over them, and pushes the results into PNUTS. For example, we might store user profiles in PNUTS that influence what content we show each user. Every few hours new user click logs arrive in a Hadoop cluster. We run analysis on these logs to generate model updates, which we write to PNUTS. The concerns in this use case are similar to the bulk load case.

The scan use case is one where an application wants to run a standard Hadoop job on data stored in PNUTS, rather than HDFS. These data sets are “native” to PNUTS, such as blog posts and comments that clients randomly write. No matter how complex the Hadoop job, the application must start by scanning PNUTS to ingest the data into Hadoop.

2. BACKGROUND

There are a large number of cloud serving systems in use today, available in open-source [2, 4, 5], as hosted services [1, 7], or only within single companies [10, 9, 13]. These systems make different fundamental design decisions. For example, Cassandra and HBase use log-structured data storage lay-

ers, and so are optimized for writes. PNUTS updates data in place and optimizes for reads. These serving systems support “CRUD” workloads (record create, retrieve, update, delete), where each operation targets one or a few records. Each must support high request rates (throughput) and low response times on the order of milliseconds (latency). A fuller comparison of these systems is available in [11].

PNUTS PNUTS is fully described elsewhere [10]; we summarize key details here. PNUTS supports both hashed and ordered tables; this work focuses on ordered tables. Tables are replicated across geographic regions by a separate messaging layer that persists writes and replicates them to the appropriate data centers. The messaging layer is provisioned with enough storage to handle prolonged data center outages. Each PNUTS table is horizontally partitioned into *tablets*. A table “foo” containing keys from *a* to *z* might be broken into ranges *a-h*, *h-t* and *t-z*, and stored in tablets named `foo.MINKEY:h`, `foo.h:t` and `foo.t:MAXKEY`.

Within a region, PNUTS consists of *routers* and *storage units*. Routers maintain a mapping from tablet names to the local storage units, and serve as a layer of indirection between clients and storage units. Within storage units, each tablet is stored in an on-disk index. PNUTS can support a wide range of indexing technologies. Our current deployment uses B-trees; specifically, MySQL [6] with InnoDB.

Every PNUTS record is *mastered* in a particular region. Record write requests are always directed to the mapped storage unit in the master region, and sent to the durable, ordered messaging layer before returning to the client. This guarantees each update is applied to each replica in the same order. We will see in Section 5 that our snapshot bulk algorithms circumvent the messaging layer and record master-ship, and thus risk introducing inconsistency.

Geographic replication differentiates PNUTS from other, single-data center cloud serving systems. This type of replication significantly reduces client-visible latencies and provides inherent disaster recovery. Global replication is one reason Yahoo! applications opt to (mis-)use PNUTS as a hybrid system rather than use a hybrid system that leaves cross-data center replication to the application.

Hadoop Hadoop [3] is the popular open-source implementation of the Map/Reduce [12] framework. Map/Reduce lets programmers easily write highly parallel batch data processing jobs by only worrying about their job’s logic, rather than the details of parallelizing it, how to handle failures, etc. Map/Reduce workloads are very different from cloud serving workloads. They consist of batch-oriented jobs that scan entire files (e.g. extracting n-grams and counts from a document) or join large files (e.g. joining a table of user models with recent user activity logs to output a new table of models). The goals are to support a high throughput of jobs, and complete individual jobs quickly (quickly on the order of minutes or hours, rather than milliseconds).

Hadoop by default uses HDFS as its underlying data store. HDFS is a distributed file system, optimized for sequential writes and large reads. It does not support in-place updates, or provide good performance for small random reads. Hadoop extends to use data stores besides HDFS. We take this approach by connecting Hadoop to PNUTS. Systems like HBase and Cassandra have similar connectivity.

Hadoop contains two centralized components; the *namenode* tracks the locations of files in HDFS, and the *jobtracker* schedules map/reduce jobs. Hadoop also consists of many

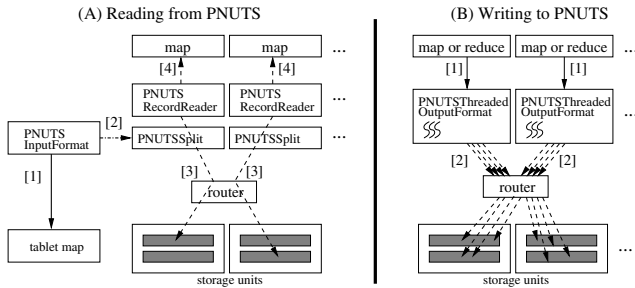


Figure 1: Hadoop-on-PNUTS.

datanode and *tasktracker* processes. These run on the same servers and can be scaled out as the number of Hadoop servers grows. Datanodes store blocks of HDFS files, while tasktrackers are responsible for running map or reduce tasks. When a Hadoop job starts, the Hadoop scheduler assigns its tasks to different tasktrackers with idle capacity. The scheduler contains some simple heuristics such as preferring to schedule a task that reads from HDFS on the same server as a datanode that stores the desired HDFS data.

Other systems: HBase, itself an open-source implementation of BigTable, is notable as a cloud serving store built directly on HDFS. Since HDFS is a write-once system, HBase stores indexes as a number of append-only differential files; we omit the details here. One attractive property of HBase is that it maintains a single data storage system, HDFS, for both serving and batch jobs.

HadoopDB [8] runs Hadoop with a cluster of standard databases (PostgreSQL) as the data storage layer. Their goal is to push as much of the Hadoop job into the database layer, where the database’s optimizer participates in processing. HadoopDB does not support serving workloads, though we can imagine it being extended to do so.

A number of “shared nothing” analytical database systems exist, and include highly-parallel bulk loading facilities that copy data from external sources into a cluster of database servers [15]. Similarly, many existing systems leverage sequential B-Tree updates in order to achieve good write throughput. One particularly sophisticated proposal uses stylized B-Tree insertions and queries to emulate log-structured indexes [14]. Standard RDBMs typically include a bulk load utility; we use MySQL’s in this work. HBase provides a bulk load utility for writing brand new tables. The programmer writes the records to be inserted in sorted order to HDFS, and then “flips a switch” that make HBase aware of the new pre-populated table.

3. ARCHITECTURE

We depict our core Hadoop-on-PNUTS architecture in Figure 1. We add PNUTS implementations of the Hadoop interfaces that read from and write to data stores. We use PNUTS’ range scan API to read data, and divide the work of scanning an entire PNUTS table by assigning an equal number of tablets to each map task. This parallelizes disk I/O, and since PNUTS maintains roughly evenly-sized tablets, divides the work reasonably well. Unlike HDFS on Hadoop, we do not co-locate map tasks with the input tablets. Instead, map tasks issue scan requests through the PNUTS router, like any other PNUTS client. Figure 1A illustrates reading from PNUTS.

Naively writing to PNUTS is even simpler. For each

Hadoop record we want to load, we make a REST-ful HTTP request to PNUTS. The problem with this approach is that PNUTS insert latency is in the 10s of ms, much longer than the time to write to HDFS. PNUTS can handle a lot of parallel inserts, however, and doing serial inserts underutilizes PNUTS’ capacity. One way to increase this parallelism is to instantiate many Hadoop tasks, and have each perform one-at-a-time inserts. The number of tasks required to saturate PNUTS would be quite large, however, incurring high overhead cost. Instead, we extend the *OutputFormat* Hadoop abstract class with *PNUTS Threaded OutputFormat*, which immediately queues write requests and returns. A pool of worker threads pull records off the queue and make HTTP requests to PNUTS. We tuned the number of worker threads to maximize throughput, allowing a relatively small number of Hadoop tasks to reach PNUTS’ saturation point. To avoid impacting concurrent serving workloads, we simply throttle the threads to a specific target throughput.

A major advantage of this scheme is that once we plug in our extended PNUTS access classes, running map/reduce jobs that manipulate PNUTS data is no harder than running ones that access HDFS. Moreover, existing applications written for HDFS can be run over PNUTS with trivial changes.

For our conversion from legacy system L to PNUTS, we snapshotted a table from L and stored it in HDFS. In this case, then, we read from HDFS, as is default in Hadoop, but wrote to PNUTS. Since our conversion simply reads records from the snapshot and writes them to PNUTS, we run a map-only job. This is similar to Figure 1B. In contrast, other use cases may do very sophisticated analyses in Hadoop, including reduce, and perhaps multiple map/reduce jobs. Since Hadoop lets us cleanly separate the plumbing for reaching PNUTS from application code, we just as easily handle complicated Hadoop applications as simple ones.

4. FAILURE HANDLING

The Hadoop-on-PNUTS system we have described so far works well in the absence of failures. Failures are common at the scale at which PNUTS and Hadoop operate, however, and we cannot rely on simple recovery schemes that force us to restart jobs and/or make PNUTS unavailable, such as dropping and recreating partially created tables.

Our L-to-PNUTS bulk load example illustrates the broad classes of failures we see in all of our hybrid use cases:

- **Failure to read input:** Hadoop may fail to parse the input data. Even when input is stored in HDFS, which is tightly coupled with Hadoop, it may be corrupted (e.g. an improperly compressed file). These failures typically impact entire files, rather than individual records.
- **Hadoop task failures:** Hadoop tasks may fail due to Hadoop errors or application bugs. These failures exist in a stand-alone Hadoop, and carry over to ours.
- **Failure to write output:** The output system (PNUTS in our bulk load) will not accept malformed write requests, which are typically caused by application bugs, or subtly incompatible record formats.
- **Network failures:** Individual requests may fail for transient reasons, (e.g. timeouts) and may succeed if re-tried. Similarly, all requests may begin to fail due to a network partition. Finally, the same symptoms may appear if we mis-configure our job (e.g. incorrect hostnames). We distinguish these variations later in this section.

One of the advantages of using Hadoop to drive bulk operations is that we inherit its failure handling. In Hadoop, failure handling takes place at the task level. If a task fails for any reason, that task gets restarted. If the task fails after a certain number of re-tries (e.g. 3), the entire job fails. We leverage this model for several of our failure cases. If a Hadoop task fails for Hadoop-specific reasons, Hadoop fails and retries the task. Likewise, if we fail to parse input data, we allow tasks to fail and be re-tried. Other failure modes do not fit well with the Hadoop model; in particular, if a task failure triggers a job failure after the job has made significant progress, we restart the job from a checkpoint instead of from the beginning (Section 4.3). Our failure handling techniques rely on the key property that writes to PNUTS are idempotent. When recovering from a failure, we can freely repeat writes without sacrificing correctness. We cannot, however, miss any writes entirely.

In the rest of this section, we describe how we combine these failure models by allowing each abstraction to “leak” into the other: OLTP-style logging and checkpointing are added to Hadoop jobs, and Hadoop-style task restarts are used to deal with large numbers of record-level failures in PNUTS. Our failure handling uses several types of logs, which we summarize here:

Log Name	Description
Retry	Individual failures formatted as in input
Detailed	Same as retry, but with debugging info
Short-circuit	Tasks ended due to widespread failure
Checkpoint	Tasks that successfully completed

4.1 Logging

The Hadoop recovery model is a poor fit for failures that involve writes to the output system. By default, upon detecting a failed record write attempt, Hadoop would simply fail and retry the entire task. This allows deterministic failures (e.g. a corrupt record) to cause entire jobs to fail, and allows rare non-deterministic failures (e.g. timeouts) to trigger the retry of entire tasks instead of single record insertions. In both cases, we need to log the failures and move on; if we detect a deterministic record failure, we report it to the user. If a failure may be non-deterministic, we retry the record insertion individually.

We introduce a simple logging framework. Each record write attempt returns success or failure. We retry each attempt a configurable number of times (e.g. 3), before declaring it a failure¹. We log each failure twice in HDFS files: once in a *retry* log, and once in a *detailed* log. For these and other logs, each task writes to its own logs, which we merge at job end. We write the failed attempt to the retry log in exactly the same format as it appears in the input, while the detailed log includes more information, such as return codes. The purpose of the retry log is to recover from transient errors, or bugs that only impact a subset of the input. Once such errors are corrected, we simply run the batch job over the retry log. This new batch job in turn creates a new retry log that is a subset of the old one. Once a job run using this technique completes without logging any errors, we have successfully processed the entire original input. Ultimately, though, we may be left with failures that will never

¹PNUTS has a variety of return codes, some of which indicate a particular request will never succeed. For such return codes, we can write to the error logs on the first attempt.

succeed (e.g. malformed records). The detailed log helps us understand the errors, and either correct these records so they are accepted, or permanently fail them.

4.2 Short-circuits

Our strategy to this point is to log individual failures, while continuing to progress with tasks and jobs. Suppose failure to reach the output system is pervasive, i.e. most or all requests fail. We do not want to blindly log every failure, and only detect that this has happened when the job ends and we see the failure logs. First, this will effectively write out the entire input to HDFS two additional times (one for each log). Second, if the batch job is long-running, we waste a lot of time to no benefit. Our solution is *short-circuiting*. Every Hadoop task maintains its count of successes and failures over a recent time window (e.g. 1 minute). If the success percentage is ever below some threshold (e.g. 90%), the task short-circuits. It instantly comes to an end, skipping whatever input remains. When a task short-circuits it records this in a *short-circuit log*. Note that a single task short-circuit does not short-circuit the entire Hadoop job. Each task must individually start, detect a high failure rate, and short circuit itself. Short-circuiting tasks still report success to Hadoop, so Hadoop does not re-try them. With a small enough window, the job still comes to an end relatively quickly. Upon job completion, the presence of a non-empty short-circuit log indicates some short-circuits have occurred. A short-circuit means we did not parse through all of the input, and so we cannot call the job a success. Instead, we must repeat part or potentially all of the job.

4.3 Checkpointing

Some short circuits are easy to live with. Suppose we misconfigure the Hadoop job such that it cannot reach the output system. Our job will quickly short-circuit; we reconfigure and restart, with minimal lost time. On the other hand, suppose we run a job for hours, only to have a network partition short-circuit it. We experienced a failure like this in testing our L-to-PNUTS application. Our PNUTS test cluster failed, causing a 15 hour job to short-circuit after 12 hours. While this seems painful enough, we consider 15 hours a short job and, in general, run jobs that take days to complete. Suppose the connection between Hadoop and the output system is unreliable, or the output system itself is unreliable. A multi-day job may never run to completion without short-circuiting. In this case, repeating a short-circuited job is at best very painful and, at worst, prevents us from ever finishing the job.

To alleviate short-circuit pain, we use *checkpointing*. When a job fails, we want to restart it from where we left off, rather than from the beginning. We achieve this using yet another log, the *checkpoint log*. Whenever a task starts, it notes the *starting point* of its input data. When the input is HDFS, for example, this is a file and offset. When the input is a table where keys are unique, as in PNUTS, this can be the first key scanned. When a task completes successfully, it logs this starting point to the checkpoint log (note that every task must log *either* a checkpoint or a short-circuit). Suppose now some job short-circuits before completion. We restart the job with the same number of tasks, and the input divided identically among them. We also pass the failed job’s checkpoint log as input to the new attempt. As each task starts, it determines its starting point as usual, but

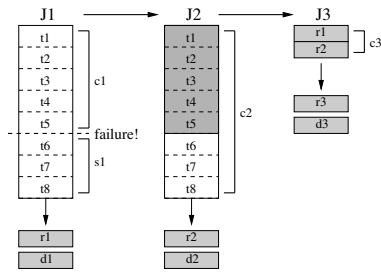


Figure 2: Log usage example.

now immediately also searches the checkpoint log to see if it contains its starting point. If its starting point is present, this indicates the previous incarnation of this task completed successfully, and we can skip it here. The task checkpoints itself and ends (from an implementation standpoint, this is nearly identical to short-circuiting). This ensures that, in the worst case, widespread batch job failures only cause us to repeat the work of tasks in progress when the failure occurred. The upper bound on the total amount of repeated work is the product of the number of parallel tasks and task length. Just as in traditional Hadoop, it is good practice to keep the unit of failure small by running lots of small tasks instead of a few long ones.

We add some key comments on checkpointing. First, we repeat writes to PNUMS, and only try to minimize the amount of repeated work. Second, we propagate checkpoint logs across multiple failed job attempts; once a task completes, we do not re-run it. We also must propagate failed jobs’ retry logs. A task does not successfully complete within a failed job unless all its input records are successfully written to the output system *or* are logged as individual failures.

While the above approach covers map-only jobs like our L-to-PNUMS conversion, we can generalize it to handle jobs with reduce tasks. Completed reduce tasks log the hash or range partition of data they are handling to the checkpoint log. On restart, we must repeat all map tasks, but avoid repeating completed reduce tasks. Alternatively, if maps do a lot of processing and dominate running time, we can write map output to HDFS, and run a subsequent job over that.

4.4 Example

We cover three failure handling features: failure logging, short-circuiting, and checkpointing. Figure 2 presents a simple example showing how these features combine, using the L-to-PNUMS case. We have a Hadoop cluster capable of running 4 parallel map tasks, and an input file containing 8 million records. We run 8 total tasks $\{t_1, t_2, \dots, t_8\}$. Job J_1 experiences a widespread failure about halfway to completion. It produces retry log r_1 , detailed log d_1 , checkpoint log c_1 and short-circuit log s_1 . $c_1 = \{t_1, t_2, \dots, t_5\}$ and $s_1 = \{t_6, t_7, t_8\}$. We propagate the logs forward to a second job attempt J_2 , which does not re-run the tasks in c_1 , successfully completes the tasks in s_1 , and produces retry log r_2 and detailed log d_2 . We are now left with two retry logs to process, r_1 and r_2 . We run a third job J_3 over just $(r_1 + r_2)$. Some failures now succeed, but we are left with a final retry log r_3 and detailed log d_3 . We report these errors.

5. IMPROVING BATCH JOB SPEED

Hadoop provides our bulk load task with more parallelism and throughput than PNUMS’s standard write path is able

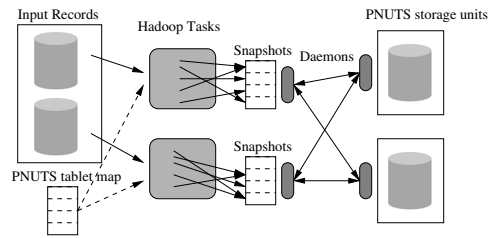


Figure 3: Snapshot import architecture.

to handle. Across a range of workloads, we (and others) have observed that, if we limit ourselves to the techniques in Section 3, then one or two Hadoop nodes running a few tasks can saturate a PNUMS cluster with ten storage units.

This section addresses two types of overhead that are incurred by PNUMS and avoided by HDFS:

Synchronous updates Each client connected to PNUMS sends a single request at a time, and waits for a response before sending the next request. The parallelization techniques discussed in Section 3 reduce, but do not eliminate this overhead.

Large working sets HDFS uses sequential writes to ensure a small fraction of the data on disk is being updated at a time. In contrast, the techniques in Section 3 let Hadoop jobs write data in arbitrary orders to arbitrary locations on disk. Because the updates have no intrinsic locality, the PNUMS storage units spend most of their time paging data to and from disk.

Note that these sources of overhead are intrinsic to serving workloads. If updates must be processed durably and with low latency, then PNUMS has no choice but to synchronously apply updates in whatever order the application chooses. Therefore, any approach that entirely avoids these issues would be inappropriate for serving workloads². In this section we introduce such a batch update mechanism. It asynchronously updates PNUMS tables and reorders writes for good locality. It does so by circumventing the normal PNUMS write path, and therefore has major consistency implications. We discuss consistency in detail in Section 6, and evaluate the performance of our new technique in Section 7.

This section focuses on improving batch write performance, and not batch read performance. For batch reads, we use the PNUMS range scan API, which uses sequential I/O to perform scans when possible. Section 7 shows scan performance is reasonably good.

We call our batch-optimized bulk import jobs *snapshot jobs*. Instead of using the standard PNUMS API, we generate *snapshot* text files suitable for direct import into PNUMS’ underlying MySQL tables. PNUMS maintains a 1-to-1 mapping of tablets to MySQL tables, allowing us to use MySQL’s bulk load mechanism (note we freeze the set of tablets while importing). For example, a snapshot job might issue the following MySQL request:

```
LOAD DATA INFILE foods.apple:banana.m3.20.txt
REPLACE INTO TABLE foods.apple:banana
```

This loads records from a snapshot text file into the MySQL table assigned to the “apple-to-banana” tablet of the PNUMS

²Techniques such as write-optimized log-structured indexes and light-weight commit protocols reduce these overheads. None eliminates them entirely.

“foods” table. The snapshot file is the 20th snapshot generated by map task number 3. Since we may generate many snapshots per tablet during a batch job, we must uniquely name them.

The snapshot may only contain records that map to this tablet; if we insert other records, such as “cherry,” into the tablet’s MySQL table, Pnuts will not correctly find them later. Each snapshot file contains 1 record per line, formatted to match Pnuts’ internal MySQL schema. Note that we use MySQL’s “replace” semantics; if we load a record that already exists, we overwrite it, rather than reject the new record. This ensures that bulk imports are idempotent.

Finally note that, while MySQL does not require files for import to be sorted by key, sorting does affect performance. We discuss this later in this section when we introduce our 3 snapshot implementations.

Figure 3 presents our architecture for moving snapshots from Hadoop to Pnuts, and importing them into Pnuts. Algorithm 2 gives the general snapshot algorithm.

Algorithm 2 General snapshot algorithm.

1. Pass the Pnuts tablet map to each (snapshot writing) task. Depending on algorithm, each task either writes snapshots for every Pnuts tablet, or for a subset of Pnuts tablets assigned to the task. Each task opens a single snapshot file for each tablet to which it writes. Each snapshot is named for its target table and tablet, and Hadoop task name and counter.
 2. Tasks record Pnuts write requests to snapshot files. Once a snapshot reaches a size limit (e.g. 10 MB), the task closes it and makes it available for transport. The task creates a new snapshot and begins writing to it.
 3. *Snapshot daemons* handle transporting and importing snapshots. *Sender daemons* run on the Hadoop tasktracker nodes, while *receiver daemons* run on the Pnuts storage units.
 - (a) Sender daemons detect when a snapshot has been created, and transport the snapshot to the storage unit in *each* Pnuts region hosting the snapshot’s tablet.
 - (b) Receiver daemons watch for arriving snapshots. When snapshots are available, the daemon chooses a snapshot, maps it to a MySQL table, and imports it. The daemon can import a configurable number snapshots in parallel. The decision of what order to import snapshots in is algorithm-specific.
-

5.1 Snapshot Algorithms

Snapshot skips the standard Pnuts code stack above the storage layer and addresses MySQL directly. We leverage MySQL’s ability to optimize a batch of writes. The common step any snapshot algorithm must execute is to sort the records being imported. Ultimately, they must be written to disk on Pnuts in sorted order, even if sorted by one-at-a-time insertion into MySQL’s B-tree. There are several choices for where to sort, and the decision significantly impacts performance. To make it easy to differentiate our algorithms, we name them by the sort method.

When run in isolation, the MySQL import takes longer than the sort. Therefore, to minimize batch job time, we seek a snapshot algorithm that starts importing as quickly as possible after job start time and that minimizes total import time. **Import Sort**, described in Algorithm 3, sends unsorted snapshots to MySQL, and we therefore rely on MySQL for sorting. A key advantage of **Import Sort** is that it effectively pipelines the batch Hadoop job and Pnuts importing. For example, a map-only job like L-to-Pnuts,

Algorithm 3 Import Sort

1. Each Hadoop task opens a snapshot for each Pnuts tablet.
 2. Sender daemons send closed snapshots to Pnuts storage units.
 3. Whenever the import is below its parallelism level, a receiver daemon iterates through all available snapshots:
 - (a) Optional: If some available snapshot maps to the same tablet as the most recently loaded snapshot, import it.
 - (b) Else choose a random available snapshot and import it.
-

with a maximum snapshot size of 20MB, starts closing snapshots within 5 minutes of job start, allowing snapshot daemons to begin importing data.

The algorithm’s main weakness is that importing can be slow, especially on populated tables. We partly address this with the optional step of checking for snapshots mapped to the same tablet as the most recently imported snapshot, and choosing one of them over a random one. Each storage unit then effectively chooses a tablet, imports all available snapshots for it, and moves to another tablet.

We added this optimization after observing that import times were roughly constant from import to import, with one key exception. Any time we happened to repeatedly import multiple snapshots for the *same* tablet, the first import ran normally, but the subsequent imports were much faster. In a typical configuration of our snapshot loader, it takes 40 seconds to import the first snapshot, but only 1-2 seconds for the tenth. The difference in performance is due to the high price of bringing the tablet’s MySQL table into memory, and the comparably low price of updating B-Tree pages that are already cached in memory. We get the most mileage out of this optimization near the end of jobs, when many snapshots have accrued on the Pnuts storage units.

Grouping imports by tablet improves throughput significantly, but MySQL import is still the longest step in most of our batch jobs. Our next set of optimizations speculate that paying to produce sorted and longer snapshots will reduce MySQL import time enough to improve the overall job time. The intuition is that writing to MySQL sequentially will lead to less B-tree rebalancing, and less swapping of pages in and out of memory.

Algorithm 4 Hadoop Sort

1. The Hadoop job ends with a reduce stage, with each Pnuts tablet mapped to a single reduce task. A custom partitioner shuffles each record to the appropriate reduce task.
 2. The reduce tasks output *sorted* snapshots. Once closed, sender daemons transport them to Pnuts storage units.
 3. Receiver daemons behave identically to Algorithm 3.
-

Hadoop Sort, in Algorithm 4, uses Hadoop to execute this sort, specifically by adding custom shuffle and reduce stages to the end of every job. These stages group records by Pnuts tablet and write them to snapshots in sorted order. As expected, this improves the performance of MySQL import, but introduces a delay between the start of the job and the beginning of MySQL import (reduces cannot produce output until all maps have completed), leaving the Pnuts storage units idle for longer at job start.

Vanilla Sort, described in Algorithm 5, combines the ad-

Algorithm 5 Vanilla Sort

1. Each Hadoop task opens a snapshot for each PNUTS tablet.
 2. Sender daemons sort closed snapshots.
 3. Sender daemons transport snapshots to PNUTS storage units.
 4. Whenever the import is below its parallelism level, a daemon chooses the PNUTS tablet with the most available snapshots. For this tablet:
 - (a) Merge the sorted snapshots for this tablet.
 - (b) Pipe the merged output to MySQL import.
-

vantages of the previous two algorithms. As in `Import Sort`, Hadoop tasks begin emitting snapshots for import quickly. However, before transmitting them to PNUTS, we sort the snapshots using the operating system’s sort utility. At the PNUTS side, daemons opportunistically merge snapshots for the same tablet, and pipe the result to MySQL as it is produced. The merge performs a single pass over the (already sorted) input, introduces no additional disk I/O, and runs on an otherwise-idle CPU; we completely mask its execution cost. At the start of the batch job we get good pipelining by emitting snapshots quickly. At the middle and end of the job, snapshots are plentiful and we get faster import times by presenting MySQL with long, sorted runs.

5.2 Hadoop, PNUTS co-location

One key potential optimization is to place Hadoop tasks and PNUTS tablets so that snapshots are generated and imported on the same servers. This reduces the amount of data sent over the network at import time, though there is no way to avoid shipping snapshots from Hadoop to remote PNUTS regions. There are two reasons why even in the single PNUTS region case we have not applied this optimization. First, snapshot copy time is minor compared to snapshot generation and snapshot importing. Second, in production at Yahoo!, Hadoop and PNUTS are managed by separate teams on servers dedicated to each system. Co-locating Hadoop and PNUTS processes on the same servers is not currently feasible.

6. CONSISTENCY

PNUTS is a serving-optimized data store, and writes to it typically come from individual web requests. What is the impact on PNUTS’ consistency when a large number of writes suddenly arrive from a batch system like Hadoop? In this section we discuss consistency implications for allowing batch jobs, both when the batch job uses the standard PNUTS write path, or uses the snapshot path. We then detail simple policies for coping with these implications.

Standard write path: PNUTS ensures timeline consistency among its replicas using per-record mastery. All writes to a record, no matter where they originate, are ordered by the record’s master replica and propagate to all replicas via a messaging layer that preserves that ordering. When batch writes compete with normal writes, mastery and the messaging layer ensure PNUTS stays consistent (see Section 2 for details). The application may have preconceptions, however, about whether batch or normal writes should trump each other, and be confused by the outcome. We discuss approaches for minimizing these problems.

Snapshot write path: The snapshot approach circumvents record mastery and the messaging layer. Therefore, if we allow batch writes to compete with standard writes we risk putting PNUTS itself in an inconsistent state. Snapshot by itself may also introduce inconsistency. Suppose a record appears in the batch input twice, and we write each appearance to different snapshot files. The different snapshot files would necessarily be for the same PNUTS tablet, and the snapshot daemons might race to import these snapshots, leaving PNUTS with different versions of the record among its replicas. No matter the cause, we cannot tolerate forcing PNUTS into an inconsistent state. We discuss approaches for preventing inconsistency.

6.1 Standard write path

In the standard write path case our goal is to minimize effects that users see as undesirable, but which are fine from PNUTS’ perspective. We suggest three *isolation* approaches by which users can avoid these effects.

- **Table isolation:** The application maintains several PNUTS tables, allowing some to be targeted only by batch writes and others only by normal writes.
- **Record isolation:** The application allows certain records to be targeted only by batch writes and other records only by normal writes.
- **Field isolation:** This leverages PNUTS’ JSON flexible record schema, which lets applications define their own fields. Now, the application additionally designates each field for either batch or normal writes, but not both.

Suppose the user cannot leverage any of the above isolation models. The next step is to isolate data by time. The application designates some time periods when only batch writes are allowed, and others when only normal writes are allowed. Ultimately, the decisions on whether to use isolation, and how to enforce that isolation are left to the application. If the application can cope with competing batch and normal writes, then they need not worry about isolation. PNUTS cannot distinguish batch and normal writes, has no way of enforcing isolation, and does not care if competition between the write types exists.

6.2 Snapshot write path

The snapshot write path makes the consistency problem more difficult, as well as more important to PNUTS. As mentioned, snapshot introduces a second write path that may compete with the standard write path, or even compete with itself, to cause inconsistency. While the isolation techniques we introduced in the previous section are useful to applications, they are also under the applications’ control, and the correctness of PNUTS cannot rely upon them.

Our approach to maintaining consistency is to enable only one write path at a time. We either enable the PNUTS messaging layer to allow the standard write path, or enable the snapshot daemons to allow the snapshot write path. Turning off the PNUTS messaging layer makes all standard writes fail (recall from Section 2 that PNUTS’ messaging layer is also its write-ahead log). In complementary fashion, if the snapshot daemons are not running, it is easy to see from Section 5 that no batch writes recorded in snapshot files can be imported into PNUTS.

Switching between write paths is actually a 3-step process. We a) disable the current write path, b) wait for PNUTS to

reach a consistent state where all regions are identical, and c) enable the other write path. To ensure PNUTS has reached consistency after disabling the standard path, we must guarantee the messaging layer has delivered all messages in flight at disable time. To do so, we a) disable the messaging layer to all client-initiated publishes, b) publish marks in all channels of the messaging layer, c) wait to receive those marks at all PNUTS regions. The messaging layer delivers messages in the order they are published. Therefore, once a replica receives all marks, no further messages are pending.

We carry out a similar procedure to switch from the snapshot path to the standard path. Disabling snapshot and ensuring consistency requires a 2-phase process. First, a coordinating process checks whether all daemons are currently idle (i.e. not copying or importing any snapshot files). If a daemon reports as idle, that daemon is locked from becoming active while the disable process is under way. If all daemons report as idle, the coordinating process shuts down all daemons, and the disable succeeds. If all daemons are not idle, the coordinating process aborts the disable, and lets all daemons return to an active state.

We mentioned earlier that duplicate keys in the input could cause batch writes to introduce inconsistencies. For some inputs, this is not possible. For example, in the L-to-PNUTS conversion, the input is itself a database that enforces primary key uniqueness, and each record's L key is the same as its PNUTS key. If we cannot guarantee uniqueness, however, we must run a Hadoop job that shuffles on the eventual PNUTS key and writes to PNUTS from reduce. Any time duplicate keys appear, these will be handed as a set to a reduce task. The reduce code decides how to deal with this set, but must emit only a single write for that key.

7. EVALUATION

This section discusses our Hadoop-on-PNUTS experimental evaluation. While our focus is PNUTS performance, we provide some standard Hadoop-on-HDFS numbers as a baseline for comparison. Our goal is not to show that PNUTS can beat HDFS on batch jobs, but instead is to characterize the trade-off made when using PNUTS, in terms of gained serving capabilities and lost batch performance. We compare batch write algorithms, both to measure improved batch write performance and interactions with PNUTS serving traffic. We summarize our findings.

- Batch writes to PNUTS using **Standard** writes are at least an order of magnitude slower than batch writes to HDFS. This strongly motivates snapshot algorithms. The best snapshot algorithm, **Vanilla Sort**, is 4x slower than writing to HDFS, but only 33% slower than writing sorted output to HDFS.
- Scanning PNUTS takes about 4x longer than scanning HDFS. In addition, the method (**Standard** or **Snapshot**) used to load PNUTS impacts subsequent scan performance; importing sorted snapshots leads to faster scans.
- **Standard** batch writes have a deleterious effect on serving throughput and latency, which we can effectively mitigate by throttling the batch workload.
- Snapshot algorithms have almost no effect on serving throughput and latency, but we can only invoke these algorithms when standard writes are disabled.

7.1 Setup

We deployed a Hadoop 0.20.2 cluster containing 6 datanodes/tasktrackers and 1 namenode/jobtracker, and a single-region PNUTS cluster containing 6 storage units, a tablet controller and a router. We initialized PNUTS tables with 120 tablets. PNUTS runs MySQL 5.1 on its storage units. To accurately mirror how these systems are used in practice, we deployed them on separate sets of machines (no co-location). All machines are server-class (8 core 2.5 GHz CPU, 8 GB RAM, 5 disk RAID-5 array, gigabit ethernet). We configured each Hadoop tasktracker to run up to 6 map tasks and 2 reduce tasks. To leverage the RAID-ed PNUTS storage unit disks, for all batch snapshot algorithms, we set parallelism to 4. We found values of 2-5 give the fastest job completion times, while 1 or 6+ degrade performance.

To separately evaluate batch write and batch read performance, we isolated each of these within Hadoop jobs. One job synthetically generates records and inserts them into a specified output system (HDFS or PNUTS). Another reads records from the specified system (HDFS and PNUTS), but then simply drops them.

We use the Yahoo! Cloud Serving Benchmark (YCSB) [11]. YCSB is a benchmarking tool that generates synthetic serving workloads for cloud serving systems like PNUTS. It drives its workload from a large number of client threads, and includes knobs to control parameters such as read/write mix, and key popularity distribution. We use YCSB as a library within our Hadoop jobs to generate random keys and payloads. We also use it as a stand-alone executable to drive our serving workloads and report performance when evaluating the impact of batch jobs on serving.

We evaluate four PNUTS batch write approaches. These are **Standard**, and the three approaches in Section 5: **Import Sort**, **Hadoop Sort**, and **Vanilla Sort**.

7.2 Write Performance

Our first experiment loads 120 million 1KB records, generated in random order, into either an empty PNUTS table or HDFS file. Figure 4 shows the load times for different batch approaches. Loading PNUTS with **Standard** is by far the slowest, taking over 8.5 hours, for a throughput of 3837 writes/second. In contrast, HDFS loads in 914 seconds. The three snapshot algorithms are much faster than **Standard**. **Import Sort** finishes in 4790 seconds, **Hadoop Sort** in 5273 seconds, and **Vanilla Sort** is best with 4059 seconds. Recall from Section 5 that **Vanilla Sort** offers the best combination of pipelining the Hadoop job and PNUTS import, and importing large, sorted snapshots.

While the snapshot algorithms come within an order-of-magnitude of HDFS, they are still at least 4x slower. One major culprit is that to load PNUTS, we must at some point sort the records, whereas we can write records to HDFS in the order they are produced. The benefit gained from sorting is that we can use PNUTS to randomly read these records and efficiently read them in sorted order or grouped by key, while HDFS cannot efficiently support this functionality. To measure the price of sorting we ran the same Hadoop job, but now sort it with reduce tasks before writing the output. Completion time now rises to 3023 seconds, leaving **Vanilla Sort** only 33% slower. Readers familiar with systems like HBase may note that this Hadoop job mimics the bulk load utility of HBase mentioned in Section 2. Since HBase records must be sorted with HDFS files

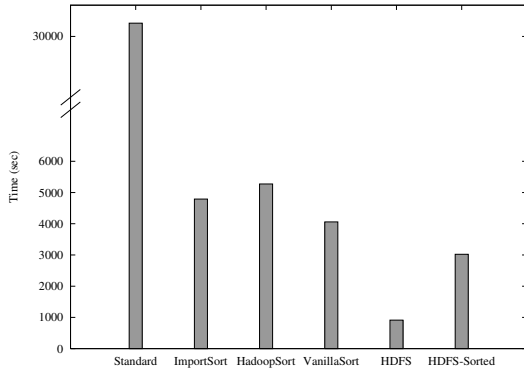


Figure 4: Load times for 120 million 1K records.

to support random reads, the utility (precisely, Hadoop jobs using the utility) must also sort input.

Our PNUTS and HDFS write experiments are apples-to-apples in the sense that both write to 6 storage nodes, whether storage units or data nodes. On the other hand, we arguably give PNUTS an advantage by using 6 Hadoop nodes *and* 6 PNUTS storage units, versus only 6 total nodes in the HDFS-only experiments. We also repeated the HDFS experiments with a 12 node Hadoop cluster. In this case HDFS load time decreased from 914 to 639 seconds, and HDFS-sorted decreased from 3023 to 1790 seconds.

Figure 5 repeats the load experiment, but now varies record size between 100B and 10KB, while holding total table size constant at 120GB. HDFS and HDFS-sorted are unaffected by record size. This is expected for HDFS, since it writes sequentially to HDFS, and performance is mostly agnostic to the number of record boundaries. The result is somewhat surprising for HDFS-sort. HDFS sorting consists of small sorts at each map task, sending records to reduce tasks, and merging at reduce tasks. Given enough tasks, the small sorts and merges are essentially free, and most of the added cost over HDFS is shuffling data to their respective reduce tasks. Therefore, run time is again dependent on total data size, rather than the number of records.

For PNUTS we plot two snapshot approaches, **Import Sort** and **Vanilla Sort**. In contrast to HDFS, PNUTS load times decrease as the number of records decreases. For all snapshot algorithms, no matter where we sort, we are ultimately bottlenecked on importing into MySQL, which imports a small number of large records faster than a large number of small records.

We modified our load experiment to be an update workload. The PNUTS table is pre-populated with 120 million 1K records and we update some percentage of them. This workload mimics one of our common batch write use cases where the table contains user profiles, and each day we update the profiles of all users who have accessed our site in the last day. We vary the number of records updated from 2.5 million (2% of the records) to 120 million (100% of the records). We plot the result in Figure 6. The trends from the initial load experiment repeat here, with a few exceptions. First, write throughput is lower than when the table is initially empty. For example, updating 120 million records with **Vanilla Sort** takes 6154 seconds, compared to 4059 seconds for initial loading. This is not surprising since writing to empty B-trees in MySQL at the start of initial load is cheaper than writing to full B-trees during all of an update

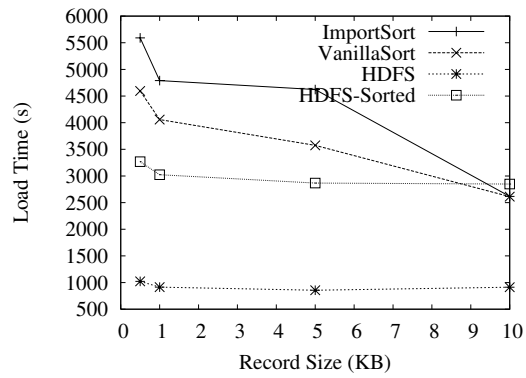


Figure 5: Load times for 120GB, varied record size.

load. Second, **Standard** actually beats **Import Sort** for the smallest update loads. Although **Import Sort** pipelines the Hadoop job and PNUTS imports efficiently, there are still a few minutes of lag from when Hadoop starts to when it begins outputting complete snapshots. For small update loads, this overhead is enough for **Standard** to beat it. Note we do not include any HDFS experiments here. HDFS files are write-once, so we cannot run an update experiment (though we could re-write the entire 120GB HDFS file).

7.3 Read Performance

We next examine read times. Our experiments compare scanning an entire PNUTS table to scanning an entire HDFS file. This mimics the use case where we run a Hadoop job over data stored in PNUTS, even if that job is only to copy the data to HDFS for repeated analyses. Figure 7 shows the results for scanning a 120GB table of 1K records. We vary the number of concurrent map tasks doing scans from 6 to 36, or from 1 to 6 per storage server (PNUTS or HDFS). We compare reading from HDFS and reading from PNUTS. For PNUTS, we compare tables populated with different load methods. As expected, the more parallel scanners for any table type, the faster the scan time. Since our disks are 5 disk RAID-5, the benefit from increased parallelism declines as we move from 4 to 6 scanners per server, and the additional scanners cause disk contention.

Our most striking observation is that the PNUTS load method has a huge impact on subsequent scan time. For 36 concurrent scan tasks, the table loaded with **Vanilla Sort** completes in 1436 seconds, while **Import Sort** and **Standard** complete in 2471 and 2673 seconds, respectively. **Hadoop Sort**, not plotted here, is similar to **Vanilla Sort**. **Vanilla Sort** and **Hadoop Sort** both import sorted snapshots (with **Vanilla Sort** importing larger snapshots as the batch job progresses), whereas **Import Sort** imports unsorted snapshots, and **Standard** is one-at-a-time. **Import Sort** and **Standard** therefore produce highly fragmented B-Trees. The subsequent scans we do in Figure 7 then perform more disk seeks, resulting in higher completion times.

Finally, we observe that scanning HDFS is faster than scanning PNUTS. For 36 concurrent scan tasks, reading from PNUTS (**Vanilla Sort**) takes 1436 seconds, compared to only 336 seconds for HDFS. Just as with the different PNUTS load methods, we suspect that the reason is index fragmentation. HDFS's append-only layout is much less likely to lead to disk fragmentation than repeated inserts into the middle of a B-Tree, even if the inserts are sorted.

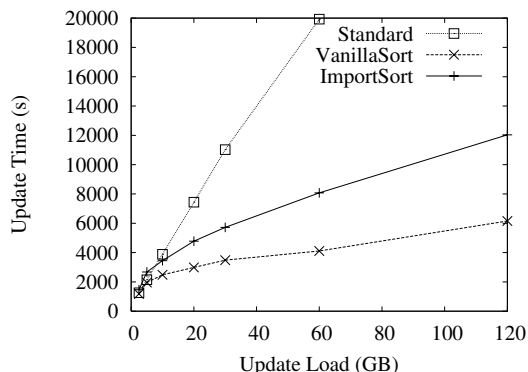


Figure 6: Varying number of updates on existing 120 million records.

Although PNUTS scans are 4x to 5x slower than HDFS (or as much as 7x to 8x for more fragmented PNUTS tables), it returns a nearly up-to-date version of a live serving PNUTS table, and that penalty may be acceptable.

7.4 Serving and Batch

We next measure the impact of batch operations on serving performance, and vice-versa, using YCSB to drive the serving workloads. We continue to measure batch completion time, and now additionally measure the key serving metrics: request throughput and request latency. We vary target throughputs for both batch and serving workloads.

Our first series runs a batch job that updates 12 million records (10% of all records) concurrent with YCSB workload “C,” which is a 100% random read workload, with keys selected with Zipfian bias. Since the workload is read-only from the serving side, we can apply both **Standard** and **Snapshot** batch jobs without introducing inconsistency, as discussed in Section 6. Figure 8 varies the target throughput of the serving workload and plots batch completion time. We plot a **Standard** batch job running unthrottled, as well as **Standard** throttled to each of 1000 operations/second and 2000 ops/second. The slower we throttle the batch job, the longer its completion time.

Vanilla Sort has similar update times to **Standard**, but does not beat it as in earlier update experiments; when the number of updates is low, as in this job, the delay to start producing snapshots is more significant. Note, however, that we can push serving throughput much higher with **Vanilla Sort** (11915 ops/second) than with **Standard** (3819 ops/second).

Figure 9 shows the same workload “C” experiment, but instead plots average serving operation latency on the y-axis. In this case, we include **Serving Only** as a baseline, where we execute the serving workload by itself. For **Standard** batch jobs, the higher the batch job’s target throughput and the higher the serving workload’s target throughput, the higher the serving latencies. At a serving target of 3000 operations/second, **Serving Only** serving latency is 6.5 ms, whereas unthrottled batch serving latency is 56 ms. We also observe that as **Standard** batch job target throughput increases, the maximum achievable serving throughput (the point at which serving latencies spike) decreases.

Vanilla Sort has a drastically different effect on serving than **Standard**. We see from Figure 8 that **Vanilla Sort** lets us reach higher serving throughputs than **Standard**.

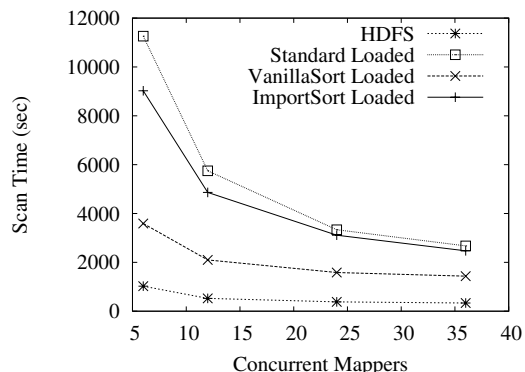


Figure 7: Scanning 120GB table, varying parallelism.

Figure 9 shows those throughputs and their corresponding latencies are similar to **Serving Only**. For example, even at 6000 operations/second, **Serving Only** latency is 7 ms, while **Vanilla Sort** is 10.3 ms. The implications of this observation cannot be overstated. For PNUTS tables that are read-only, or that can be shifted to read-only during batch update, we can run snapshot batch updates without having to overprovision PNUTS and without degrading client throughputs or latencies. This is much more attractive than either paying for additional capacity to accommodate batch jobs, or telling clients they must plan around a “fast” SLA, and a “slow” SLA for times when batch jobs are running.

Figures 10 and 11 mirror Figures 8 and 9, but using YCSB workload “A”, which is 50% read/50% update, with keys chosen randomly with Zipfian bias. Since the serving workload is not read-only and uses the standard write path, we cannot enable the second snapshot write path. Figure 10 again shows that throttling the batch job increases its completion time. Note, however, that as we increase target serving throughput, **Standard** unthrottled and throttled to 2000 ops/second converge, and both take longer to complete. At this point, the serving workload consumes enough PNUTS bandwidth that both unthrottled and throttled 2000 do not manage to perform 2000 operations/second. The **Standard** batch and serving workloads are indistinguishable to PNUTS, and compete for system resources.

Figure 11 shows the same trend as workload “A.” Throttled standard batch jobs let us achieve low serving latencies at low serving throughputs, but the latencies quickly increase with higher serving throughputs. Note the shape of the **Serving Only** plot. This is expected from a cloud serving system like PNUTS: latency is low at all throughputs, until we approach the saturation point, and requests begin to queue up in PNUTS. In practice, we provision the system to not approach saturation, so that clients always have good response times. Compare this plot to those where we run a concurrent batch job. The higher the target throughput of the batch job, the sooner and more rapidly serving latency increases (the shape of unthrottled batch is actually inverted compared to **Serving Only**). In practice, this means we cannot run unthrottled batch jobs without exposing clients to a noticeable degradation in performance. We must throttle the batch job, and provision PNUTS such that the sum of the serving workload throughput and batch target throughput is sufficiently below saturation point.

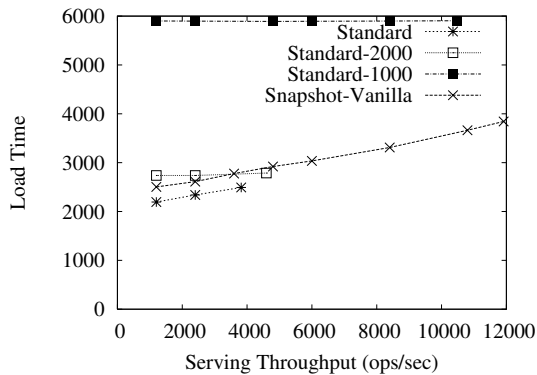


Figure 8: Update time for batch update and serving-100/0 read/write workload.

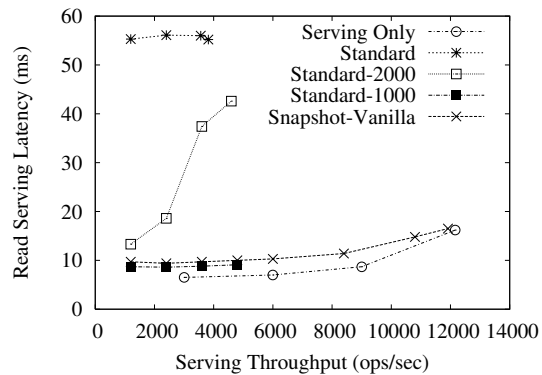


Figure 9: Serving latency for batch update and serving-100/0 read/write workload.

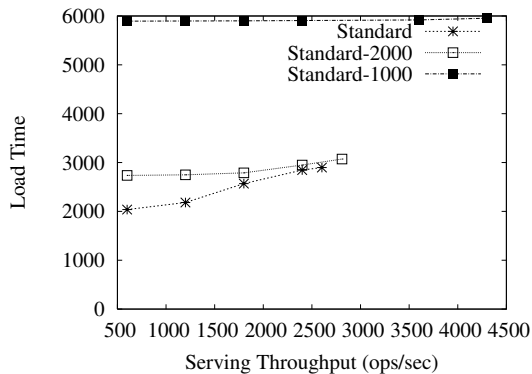


Figure 10: Update time for batch update and serving-50/50 read/write workload.

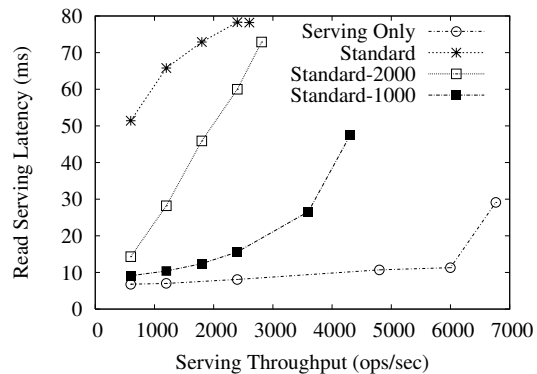


Figure 11: Serving latency for batch update and serving-50/50 read/write workload.

8. CONCLUSION

Cloud data management systems can largely be divided into batch-oriented and serving-oriented systems. At Yahoo!, for example, Hadoop and PNUTS have gained adoption for batch and serving workloads. We observe a number of applications, however, that cannot be cleanly mapped to one of these types of systems, but instead require hybrid functionality. To provide such functionality, we have combined Hadoop and PNUTS. In building this combination, we observe several key areas where the properties of each system conflict, and we alter or break the standard abstractions of batch and serving systems to mitigate these conflicts. We combine techniques from both batch and serving to enable coarse-grained and fine-grained failure detection and recovery. Serving stores give up write throughput to gain low per-write latency, and this is a bad tradeoff in the batch setting. For situations where throughput trumps latency, we have built a *snapshot* batch write path that recovers the batch-oriented performance sacrificed by PNUTS' standard write path. We show that our best snapshot algorithm, **Vanilla Sort**, is competitive with writing sorted output to HDFS. We also show that we can run batch write jobs concurrently with read-only serving workloads, with very little negative effect on serving throughput and latency. Finally, batch and serving systems have different and conflicting consistency models. We provide techniques for isolating these models from one another.

9. REFERENCES

- [1] Amazon SimpleDB. <http://aws.amazon.com/simpledb/>.
- [2] Apache Cassandra. <http://incubator.apache.org/cassandra/>.
- [3] Apache Hadoop. <http://hadoop.apache.org/>.
- [4] Apache HBase. <http://hadoop.apache.org/hbase/>.
- [5] MongoDB. <http://www.mongodb.org/>.
- [6] MySQL. <http://www.mysql.com/>.
- [7] SQL Data Services/Azure Services Platform. <http://www.microsoft.com/azure/data.mspx>.
- [8] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.
- [9] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
- [10] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.
- [11] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.
- [12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
- [13] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.
- [14] G. Graefe. Sorting and indexing with partitioned B-trees. In *CIDR*, 2003.
- [15] Greenplum database 4.0: Critical mass innovation. Technical report, EMC, 2010.
- [16] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.